



Creación de Aplicaciones Python

Lección 8: API REST con framework Flask

ÍNDICE

Lección 8: API REST con framework Flask	1
Presentación y objetivos	1
1. ¿Qué es Flask ? ¿ Por qué usarlo?.....	2
2. Creando una API REST con Flask	3
3. API REST - Iris Dataset	11
3.1. Método GET:	11
3.2. Método POST.....	13
3.3. Método PUT	15
3.4. Método DELETE.....	17
4. Puntos clave	20

Lección 8: API REST con framework Flask

PRESENTACIÓN Y OBJETIVOS

En esta lección aprenderemos a realizar un API REST usando Flask como Framework de Python, realizando los métodos HTTP más importante como son GET, POST, PUT y DELETE.



Objetivos

- Conocer el framework de Python Flask
- Realizar una API REST empleando Flask.

1. ¿QUÉ ES FLASK ? ¿ POR QUÉ USARLO?

Flask es un framework minimalista escrito en Python que permite crear aplicaciones web rápidamente y con un mínimo número de líneas de código. Está basado en la especificación WSGI de Werkzeug y el motor de templates Jinja2 y tiene una licencia BSD.

La palabra “micro” no significa que sea un proyecto pequeño o que nos permita hacer páginas web pequeñas sino que al instalar Flask tenemos las herramientas necesarias para crear una aplicación web funcional pero si se necesita puede añadir una nueva funcionalidad hay un conjunto muy grande de extensiones (plugins) que se pueden instalar con Flask que le van dotando de funcionalidad.

Flask es un “micro” Framework: Para desarrollar una App básica o que se quiera desarrollar de una forma ágil y rápida Flask puede ser muy conveniente, para determinadas aplicaciones no se necesitan muchas extensiones y es suficiente.

¿Por qué usar Flask?

- I. Incluye un servidor web de desarrollo: No se necesita una infraestructura con un servidor web para probar las aplicaciones sino de una manera sencilla se puede correr un servidor web para ir viendo los resultados que se van obteniendo.
- II. Tiene un depurador y soporte integrado para pruebas unitarias.
- III. Es compatible con wsgi: Wsig es un protocolo que utiliza los servidores web para servir las páginas web escritas en Python.
- IV. Buen manejo de rutas: URI.
- V. Soporta de manera nativa el uso de cookies seguras.
- VI. Se pueden usar sesiones.
- VII. Flask no tiene ORMs: Pero se puede usar una extensión.
- VIII. Sirve para construir servicios web (como APIs REST) o aplicaciones de contenido estático.
- IX. Flask es Open Source y está amparado bajo una licencia BSD.
- X. Buena documentación, código de GitHub y lista de correos.

2. CREANDO UNA API REST CON FLASK

Creamos el proyecto:

```
mkdir flask
```

```
cd flask
```

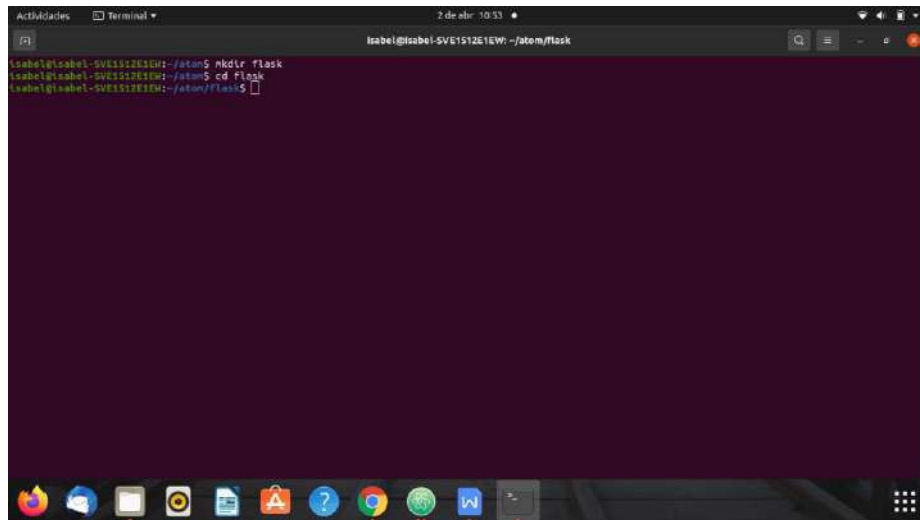


Figura 2.1. Creación de la carpeta dónde crearemos la aplicación

Una vez creado crearemos el entorno virtual donde vamos a trabajar:

```
virtualenv flaskEnv
```

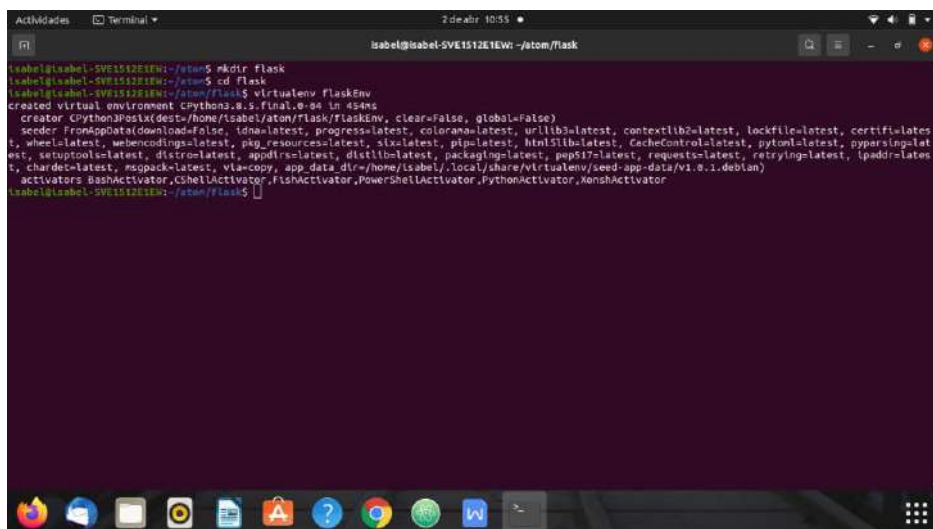
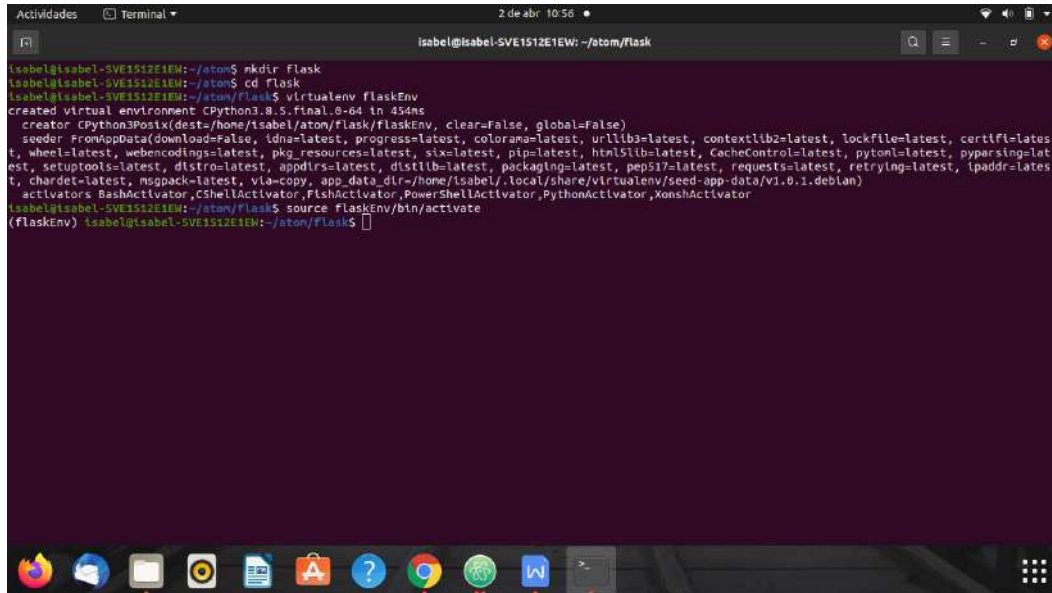


Figura 2.2. Creación del Entorno Virtual

Entramos en el entorno:

```
source flaskEnv/bin/activate
```

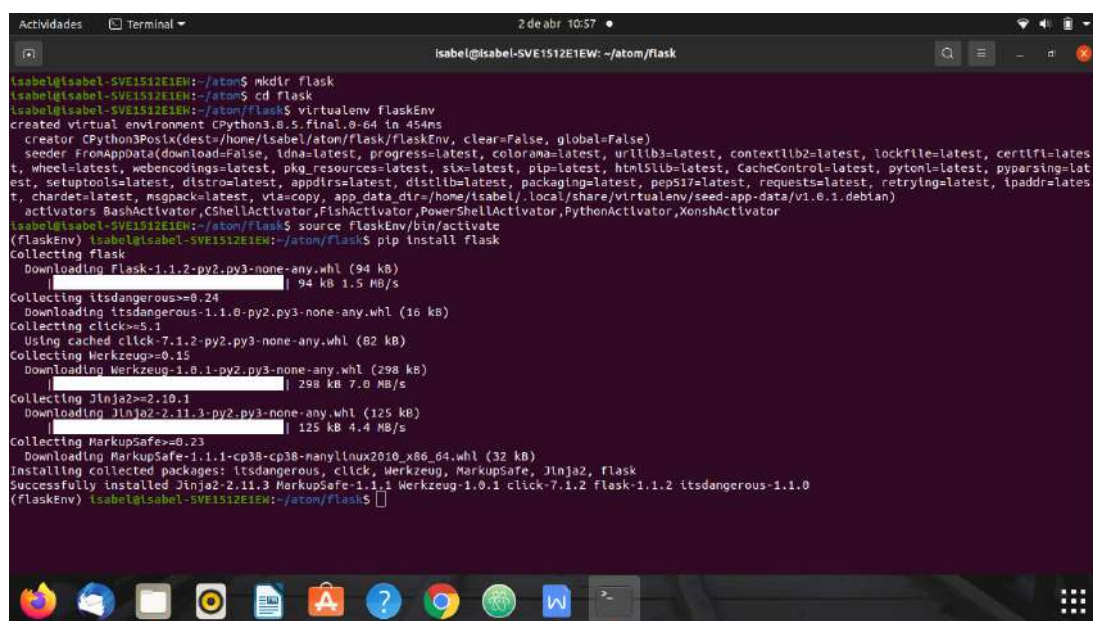


```
isabel@isabel-SVE1512E1EW: ~/atom/Flask
isabel@isabel-SVE1512E1EW:~/atom$ mkdir flask
isabel@isabel-SVE1512E1EW:~/atom$ cd flask
isabel@isabel-SVE1512E1EW:~/atom/flask$ virtualenv flaskEnv
created virtual environment CPython3.8.5.final.0-64 in 454ms
  creator CPython3Posix(dest=/home/isabel/atom/flask/flaskEnv, clear=False, global=False)
  seeder FromAppData(download=False, idna=latest, progress=latest, colorama=latest, urllib3=latest, contextlib2=latest, lockfile=latest, certifi=late
t, wheel=latest, webencodings=latest, pkg_resources=latest, six=latest, pip=latest, html5lib=latest, CacheControl=latest, python1=latest, pyparsing=lat
est, setuptools=latest, distro=latest, appdirs=latest, distlib=latest, packaging=latest, pep517=latest, requests=latest, retrying=latest, ipaddr=late
t, chardet=latest, msgpack=latest, via=copy, app_data_dir=/home/isabel/.local/share/virtualenv/seed-app-data/v1.0.1.debian)
  activators BashActivator,CShellActivator,FishActivator,PowerShellActivator,PythonActivator,XonshActivator
isabel@isabel-SVE1512E1EW:~/atom/flask$ source flaskEnv/bin/activate
(flaskEnv) isabel@isabel-SVE1512E1EW:~/atom/flask$
```

Figura 2.3. Activación del Entorno Virtual

Instalamos la librería de flask:

```
pip install Flask
```



```
isabel@isabel-SVE1512E1EW:~/atom/Flask$ pip install flask
Collecting flask
  Downloading flask-1.1.2-py2.py3-none-any.whl (94 kB)
    | 94 kB 1.5 MB/s
Collecting itsdangerous>=0.24
  Downloading itsdangerous-1.1.0-py2.py3-none-any.whl (16 kB)
Collecting click>=5.1
  Using cached click-7.1.2-py2.py3-none-any.whl (82 kB)
Collecting Werkzeug>=0.15
  Downloading Werkzeug-1.0.1-py2.py3-none-any.whl (298 kB)
    | 298 kB 7.0 MB/s
Collecting Jinja2>=2.10.1
  Downloading Jinja2-2.11.3-py2.py3-none-any.whl (125 kB)
    | 125 kB 4.4 MB/s
Collecting MarkupSafe>=0.23
  Downloading MarkupSafe-1.1.1-cp38-cp38-manylinux2010_x86_64.whl (32 kB)
Installing collected packages: itsdangerous, click, Werkzeug, MarkupSafe, Jinja2, flask
Successfully installed Jinja2-2.11.3 MarkupSafe-1.1.1 Werkzeug-1.0.1 click-7.1.2 flask-1.1.2 itsdangerous-1.1.0
(flaskEnv) isabel@isabel-SVE1512E1EW:~/atom/flask$
```

Figura 2.4. Instalación de la librería Flask

Una vez creado lo abrimos en el IDE Atom:

New window --> Add Project Folder --> atom/flask --> Aceptar

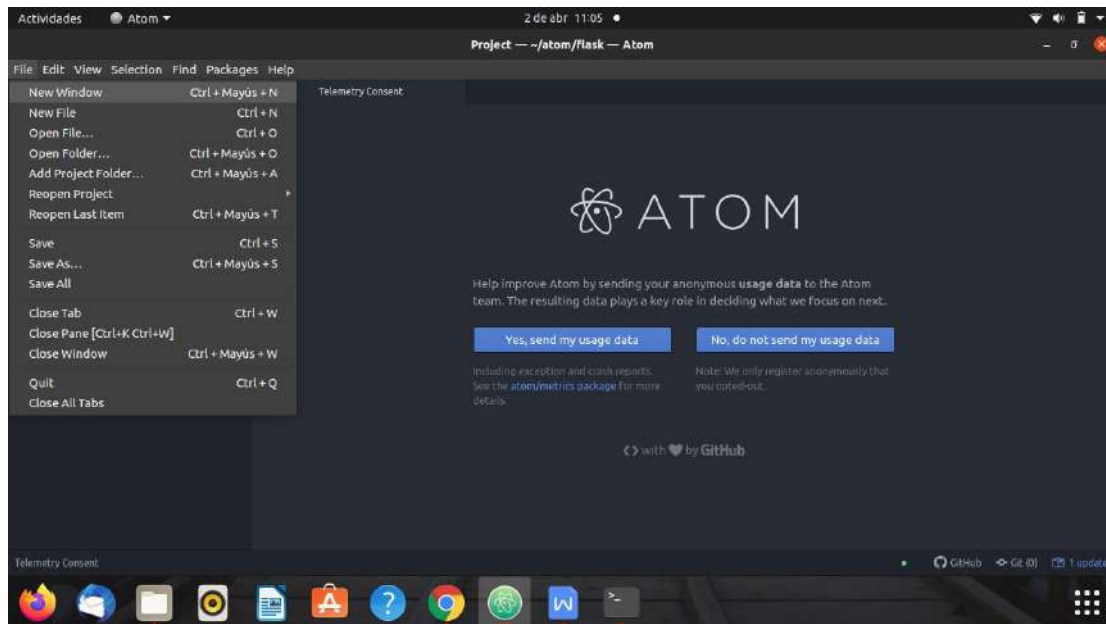


Figura 2.5. Añadir el proyecto a IDE Atom

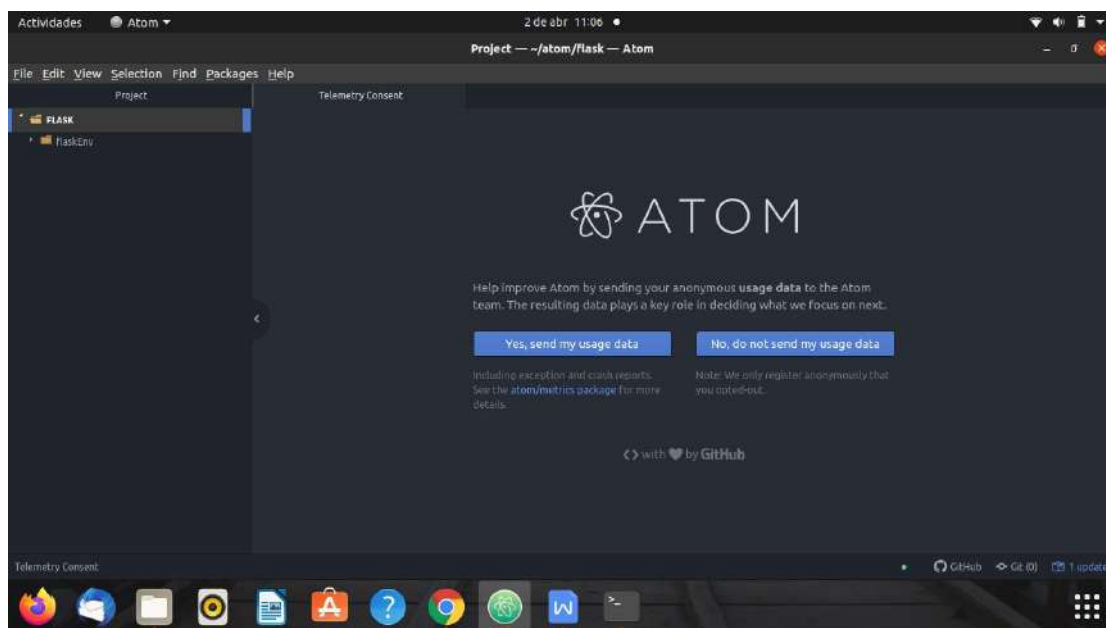


Figura 2.6. Proyecto abierto en Atom

Una vez que lo tenemos crearemos las carpetas:

Static para añadir el CSS e imágenes si es necesario.

Templates donde se encontraran las plantillas HTML

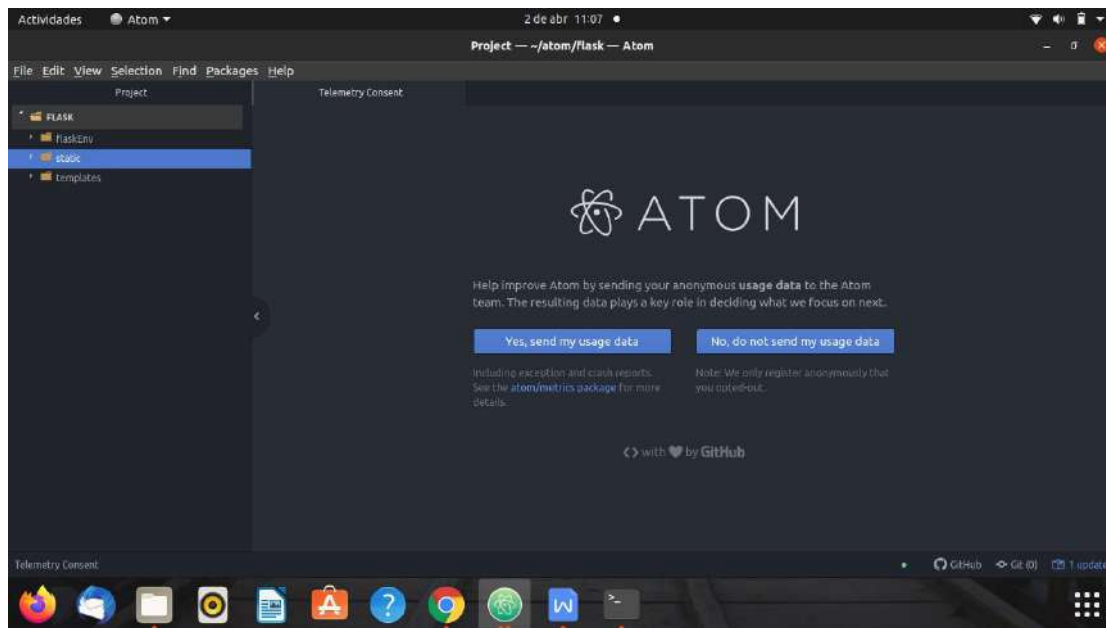


Figura 2.7. Creación de las carpetas static y templates

Nosotros en este ejemplo vamos a realizar sólo el Back-end, por lo tanto estas carpetas las creamos pero no las vamos a usar.

Creamos la aplicación Flaskr:

```
mkdir flaskr
```

Creamos nuestra primera aplicación flaskr en el archivo `__init__.py`:

```
# Librerías para crear la aplicación y recoger los datos:
from flask import Flask, request

# Creamos nuestra primera aplicación llamada app.
# Creamos "__name__" como función principal
app = Flask(__name__)

# Ruta de inicio "/", metodo GET
@app.route('/', methods=['GET'])
def home():
    return 'Bienvenido a Flask'--> Mensaje que aparecerá en la ventana

if __name__ == '__main__':
    # Correr la aplicación inicializada
    app.run(debug=True)
```

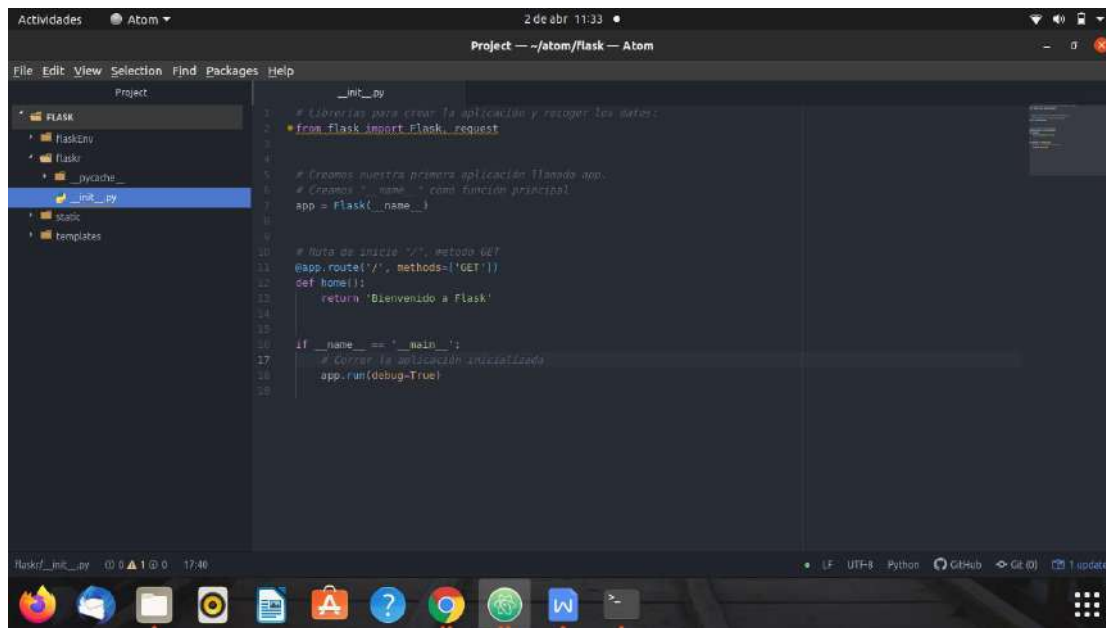



Figura 2.8. Creación del archivo `__init__.py` donde crearemos la aplicación con Flask

Una vez creada necesitamos que Flask se ejecute en desarrollo no en producción para que nos muestre todos los errores, para ello dependiendo del sistema operativo usaremos unas instrucciones u otras, en el ejemplo es Ubuntu:

Para Linux y Mac:

```

$ export FLASK_APP=flaskr
$ export FLASK_ENV=development
$ flask run

```

Para Windows cmd, use set para exportar:

```

> set FLASK_APP=flaskr
> set FLASK_ENV=development
> flask run

```

Para Windows PowerShell, use \$env: para exportar:

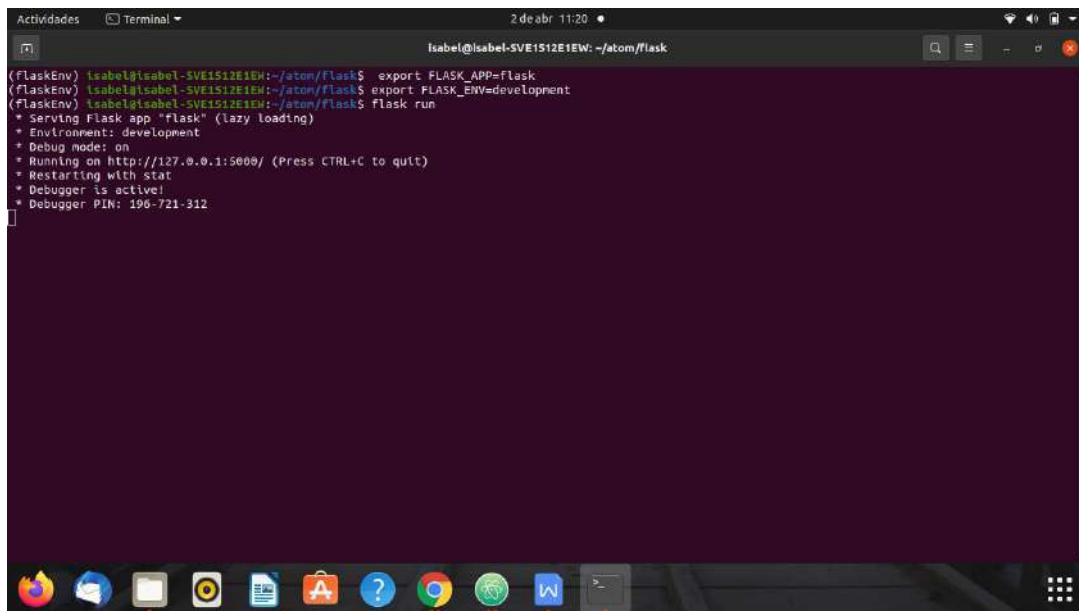
```

> $env:FLASK_APP = "flaskr"
> $env:FLASK_ENV = "development"
> flask run

```

Mostrará algo similar a esto:

- * Serving Flask app "flaskr"
- * Environment: development
- * Debug mode: on
- * Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
- * Restarting with stat
- * Debugger is active!
- * Debugger PIN: 855-212-761



```
Actividades Terminal 2 de abr 11:20
isabel@isabel-SVE1512E1EW: ~/atom/flask
(FlaskEnv) isabel@isabel-SVE1512E1EW:~/atom/flask$ export FLASK_APP=flask
(FlaskEnv) isabel@isabel-SVE1512E1EW:~/atom/flask$ export FLASK_ENV=development
(FlaskEnv) isabel@isabel-SVE1512E1EW:~/atom/flask$ flask run
* Serving Flask app "flask" (lazy loading)
* Environment: development
* Debug mode: on
* Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
* Restarting with stat
* Debugger is active!
* Debugger PIN: 190-721-312
```

Figura 2.9. En la imagen se muestra cómo se ejecuta la aplicación en desarrollo.

Esto cada vez que iniciemos el entorno tenemos que ejecutarlo.

Nota: otra manera de ejecutar la aplicación es poniendo `python <namefile>.py` pero de esta forma no nos avisará de los errores (bugs).

Vamos a la url `http://127.0.0.1:5000/`, nos mostrará el mensaje Bienvenido a Flask que hemos definido en la función:

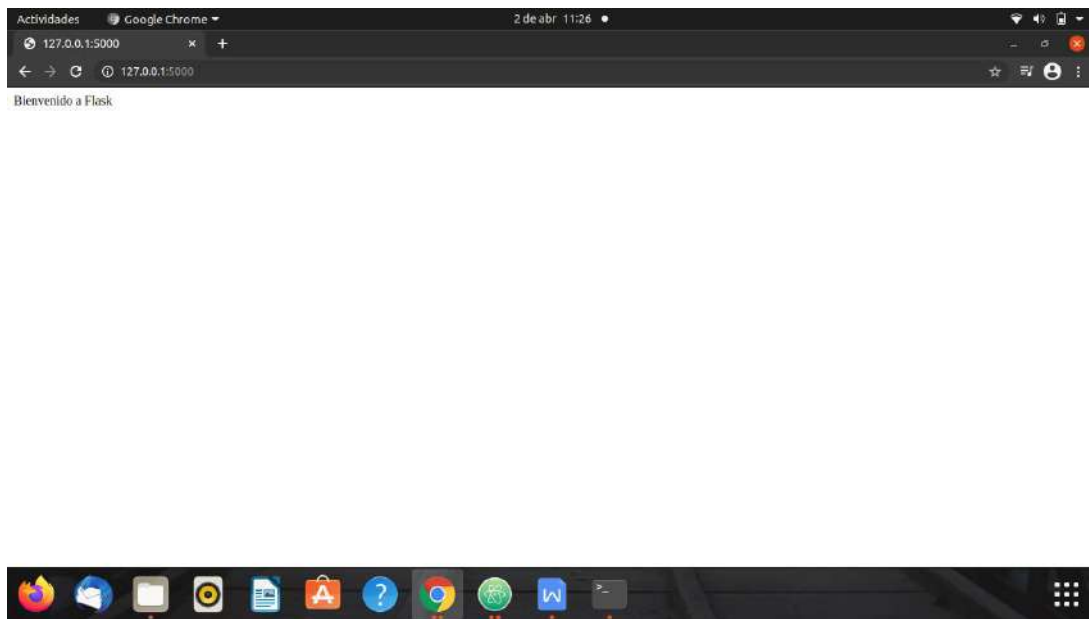


Figura 2.10. En la imagen se muestra cómo se ejecuta la aplicación en desarrollo.

Para crear el modelo de datos será necesario instalar la librería flask-appbuilder:

```
pip install flask-appbuilder
```

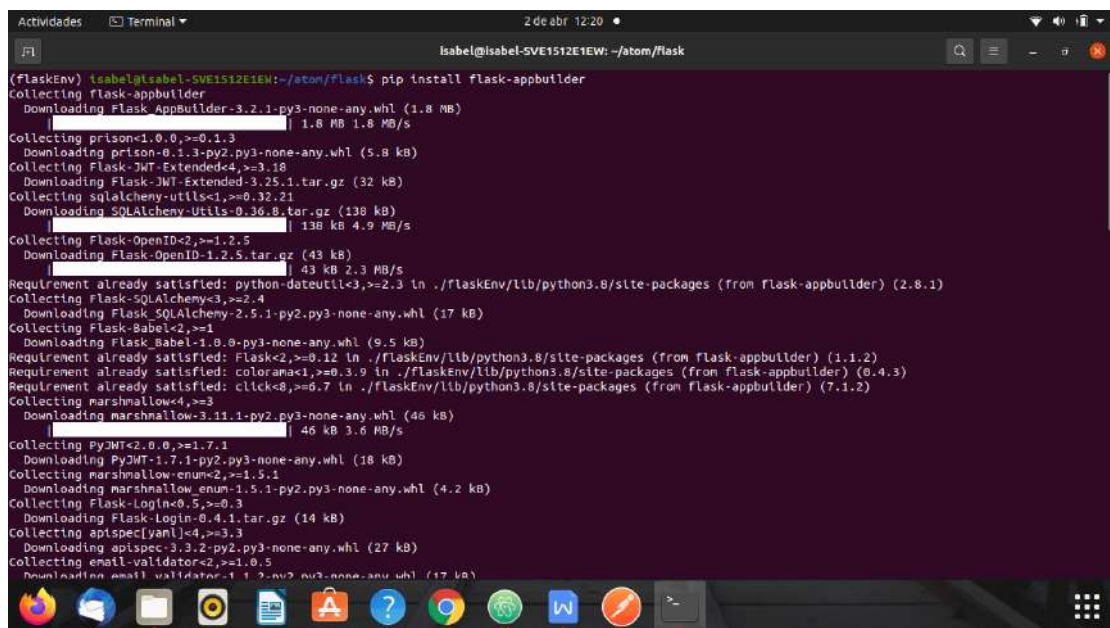
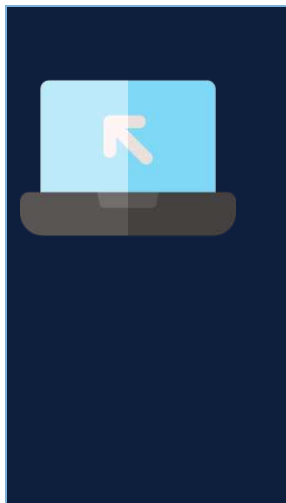


Figura 2.11. Instalación de la librería flask-appbuilder



Para más información

Nosotros no lo usaremos pero es otra manera de trabajar con Flask y similar a Django, ya que contiene la gestión de usuarios y traducción de la web.

<https://flask-appbuilder.readthedocs.io/en/latest/quickhowto.html>

Será necesario instalar pandas:

```
pip install pandas
```

```
Actividades Terminal 2 de abr 12:31
Isabel@isabel-SVE1512E1EW: ~/atom/Flask

(FlaskEnv) isabel@isabel-SVE1512E1EW:~/atom/Flask$ pip install pandas
Collecting pandas
  Using cached pandas-1.2.3-cp38-cp38-manylinux1_x86_64.whl (9.7 MB)
Requirement already satisfied: python-dateutil>=2.7.3 in ./flaskEnv/lib/python3.8/site-packages (from pandas) (2.8.1)
Requirement already satisfied: pytz>=2017.3 in ./flaskEnv/lib/python3.8/site-packages (from pandas) (2021.4)
Requirement already satisfied: numpy>=1.16.5 in ./flaskEnv/lib/python3.8/site-packages (from pandas) (1.20.2)
Requirement already satisfied: six>=1.5 in ./flaskEnv/lib/python3.8/site-packages (from python-dateutil>=2.7.3->pandas) (1.14.0)
Installing collected packages: pandas
Successfully installed pandas-1.2.3
(FlaskEnv) isabel@isabel-SVE1512E1EW:~/atom/Flask$
```

Figura 2.12. Instalación de la librería pandas

3. API REST - IRIS DATASET

3.1. Método GET:

Para ello en el archivo `__init__.py` importamos las siguientes librerías:

```
import pandas as pd
import json
```

Definimos la función:

```
# Ruta de inicio "/iris/", metodo GET
@app.route('/iris/', methods=['GET'])
def irisData():

    # Cargamos el dataset con ayuda de pandas:
    X_df = pd.read_csv('iris.csv')

    # Resumen del Dataset .describe():
    describe = X_df.describe().to_json(orient="index")
    describe = json.loads(describe)

    return describe
```

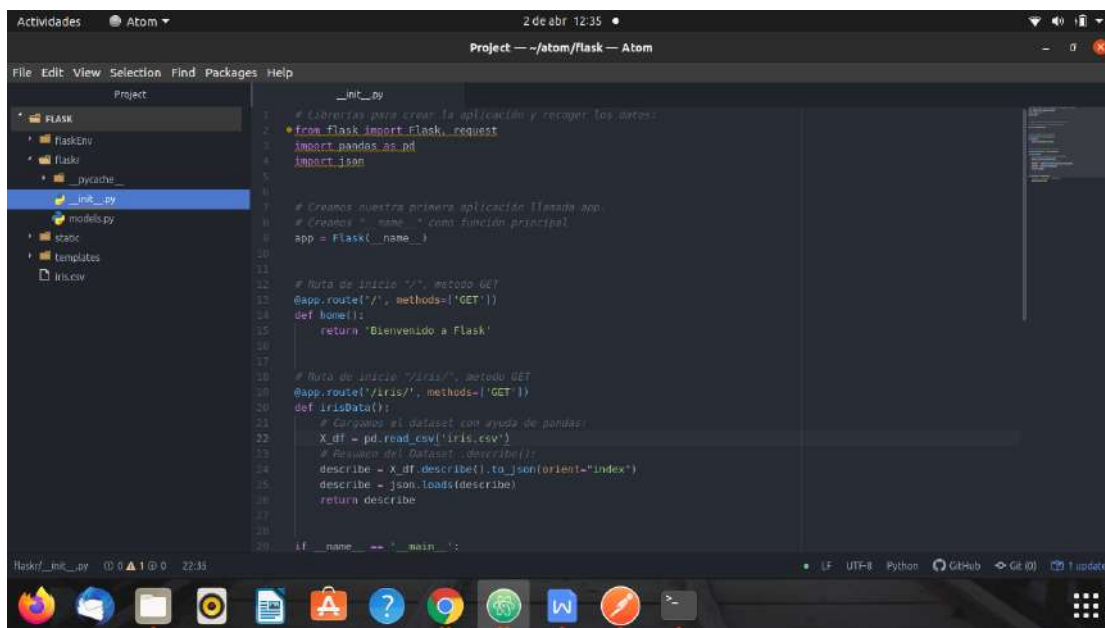


Figura 3.1. Archivo `__init__.py`

Comprobamos que funciona vamos al navegador: <http://127.0.0.1:5000/iris/>

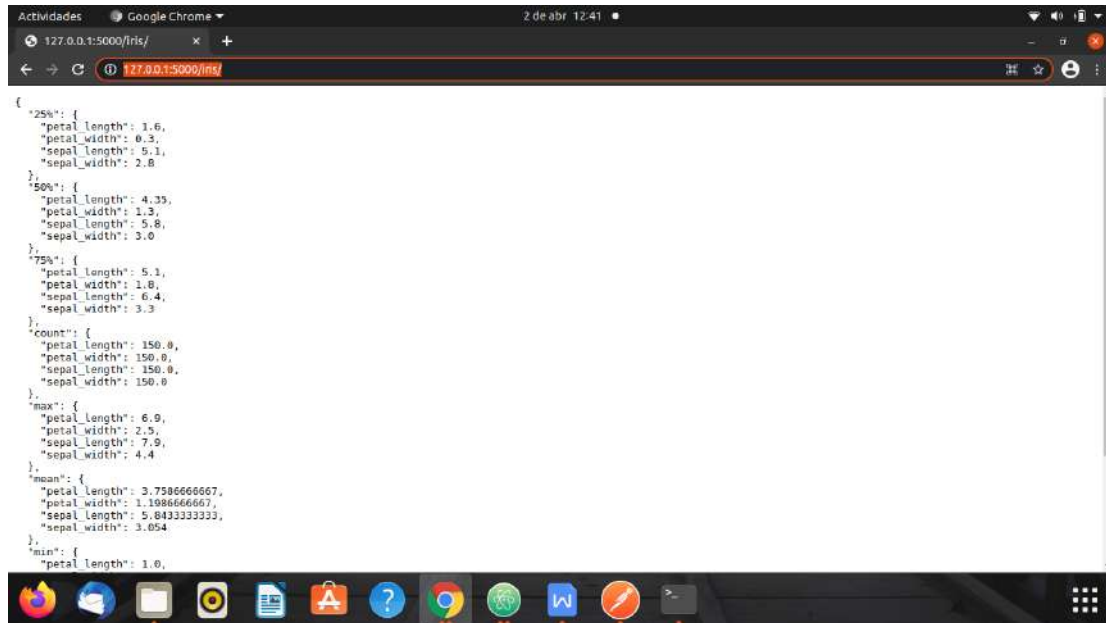


Figura 3.2. La imagen nos muestra los datos del resumen del dataset

También probamos en POSTMAN ponemos la URL <http://127.0.0.1:5000/iris/> y damos a send:

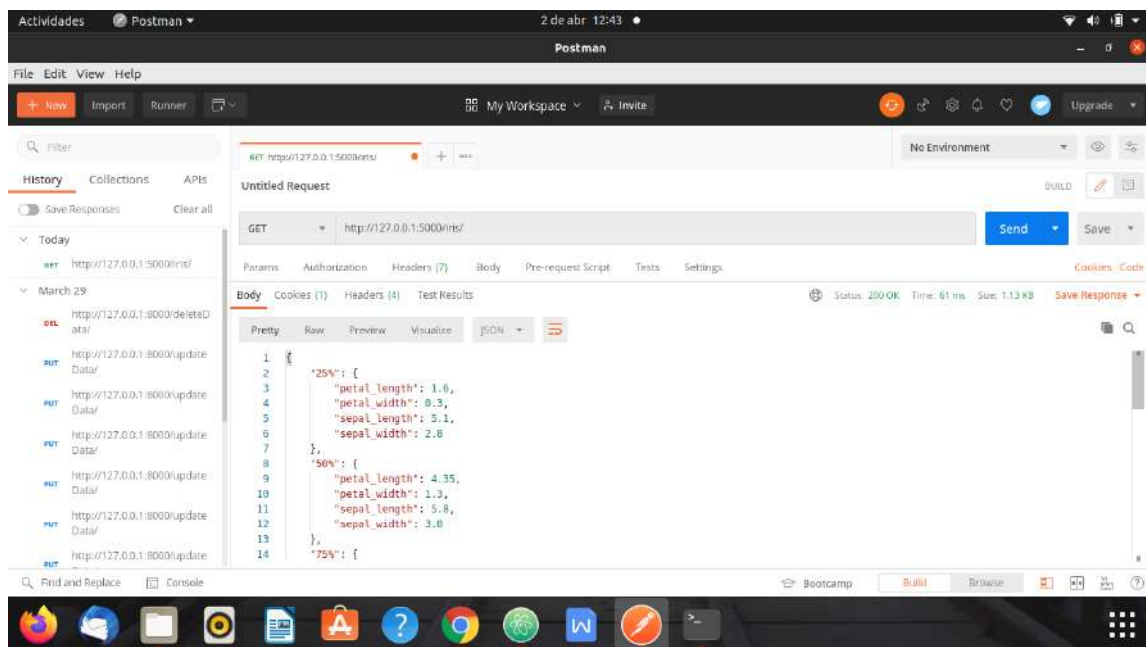


Figura 3.3. La imagen nos muestra los datos del resumen del dataset usando Postman

Nos muestra un 200 Ok y en el body de request los datos de resumen del dataset.

3.2. Método POST

Cargamos las librerías:

```
import csv

# Ruta de inicio "/insertData/", metodo GET
@app.route('/insertData/', methods=['POST'])

def insertdata():

    # definir 'data' como el conjunto de datos
    # que se reciben a través del Postman:

    data = request.data
    data = json.loads(data)

    # insertar dato en csv:

    with open('iris.csv', 'a', newline='') as csvfile:

        fieldnames = ['sepal_length', 'sepal_width', 'petal_length',
                     'petal_width', 'species']

        writer = csv.DictWriter(csvfile, fieldnames=fieldnames)

        writer.writerow({'sepal_length': data['sepal_length'],
                        'sepal_width': data['sepal_width'],
                        'petal_length': data['petal_length'],
                        'petal_width': data['petal_width'],
                        'species': data['species']})

    print("writing complete")

    return data
```

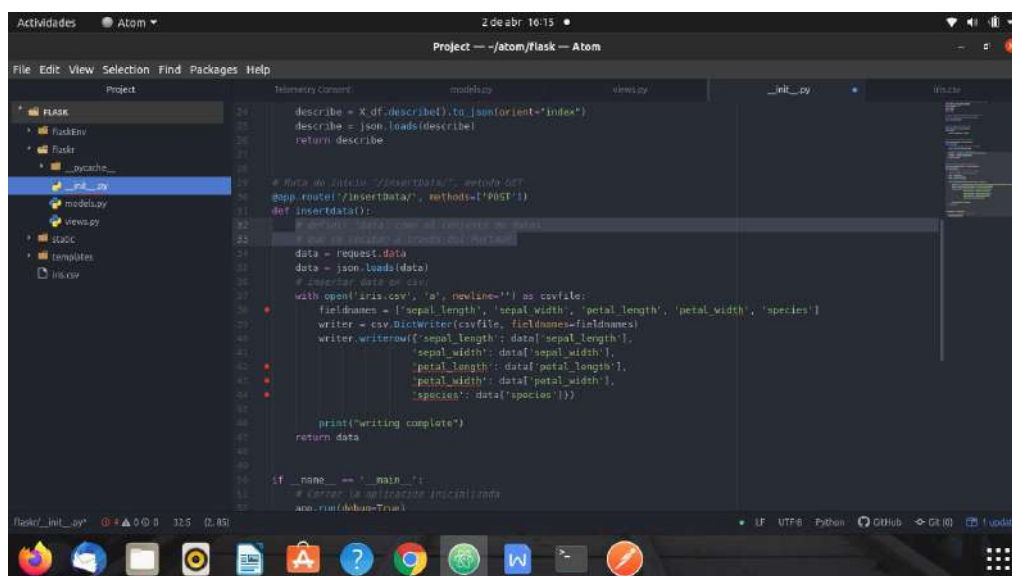


Figura 3.4. Archivo `__init__.py` con el método POST

Ponemos en el Postman POST : `http://127.0.0.1:5000/insertdata/`

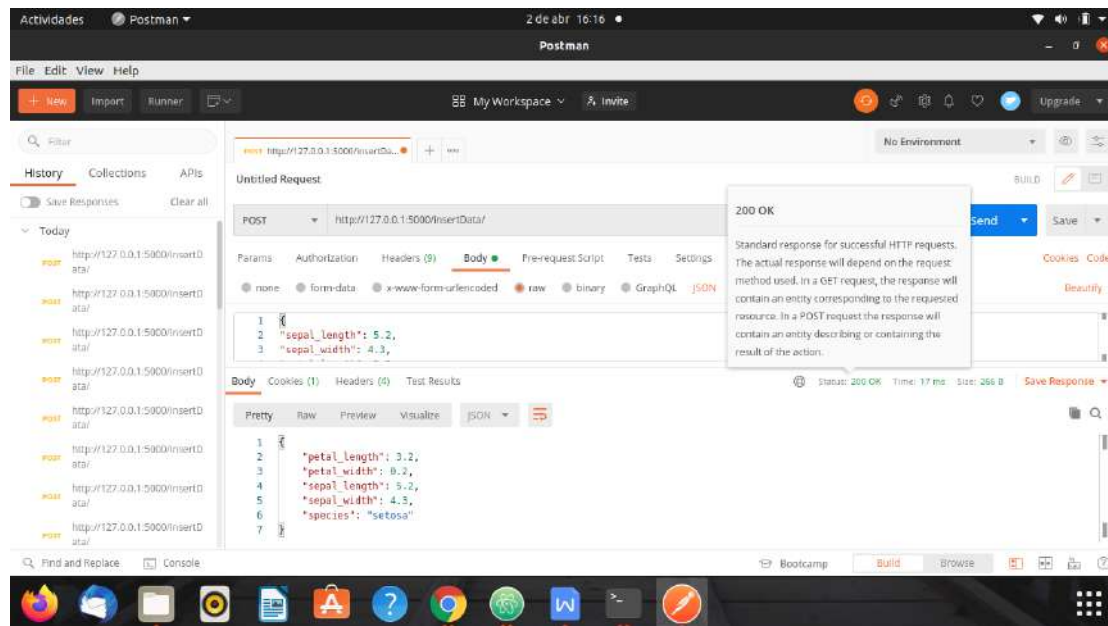


Figura 3.5. Método POST en Postman

Comprobamos en el archivo CSV que se ha insertado correctamente:

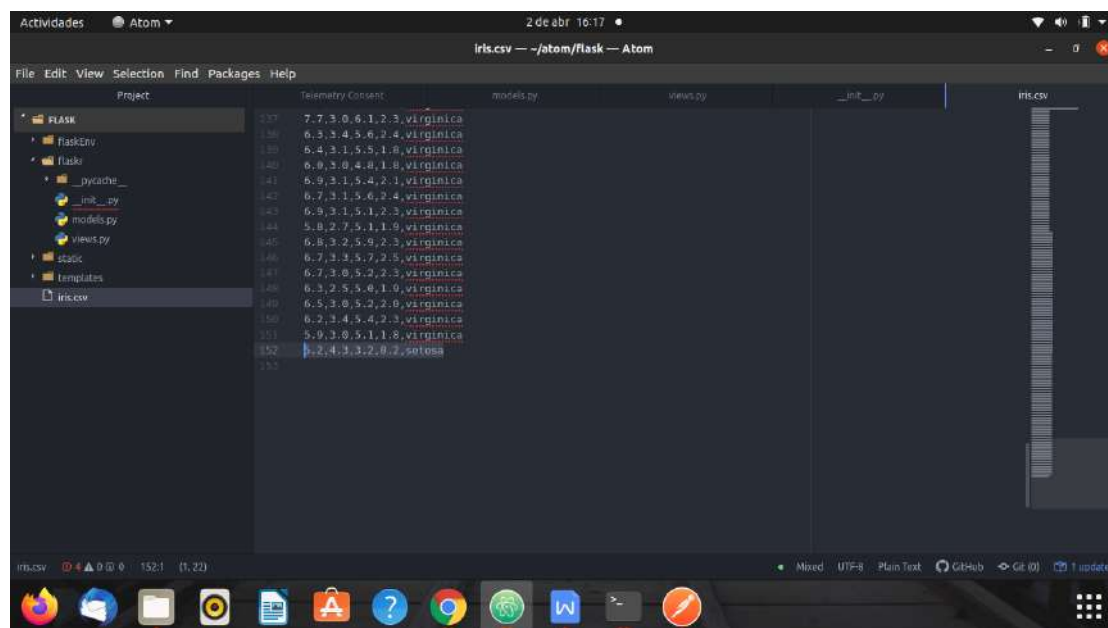


Figura 3.6. La imagen muestra el archivo CSV como se ha insertado un dato en la última fila

3.3. Método PUT

Definimos la función PUT:

```
# Ruta de inicio "/updateData/", metodo PUT
@app.route('/updateData/', methods=['PUT'])
def updatedata():
    # definir 'data' como el conjunto de datos
    # que se reciben a través del Postman:
    data = request.data
    data = json.loads(data)
    df = pd.read_csv('iris.csv')
    # sustituimos la última fila del dataset cada uno de los valores
    # con los datos que recibimos 'data':
    df.loc[df.index[-1], 'sepal_length'] = data['sepal_length']
    df.loc[df.index[-1], 'sepal_width'] = data['sepal_width']
    df.loc[df.index[-1], 'petal_length'] = data['petal_length']
    df.loc[df.index[-1], 'petal_width'] = data['petal_width']
    df.loc[df.index[-1], 'species'] = data['species']
    # convertir a csv
    df.to_csv('iris.csv', index=False)
    # mostrar el último dato en formato Json:
    result = df.iloc[-1].to_json(orient="index")
    return result
```

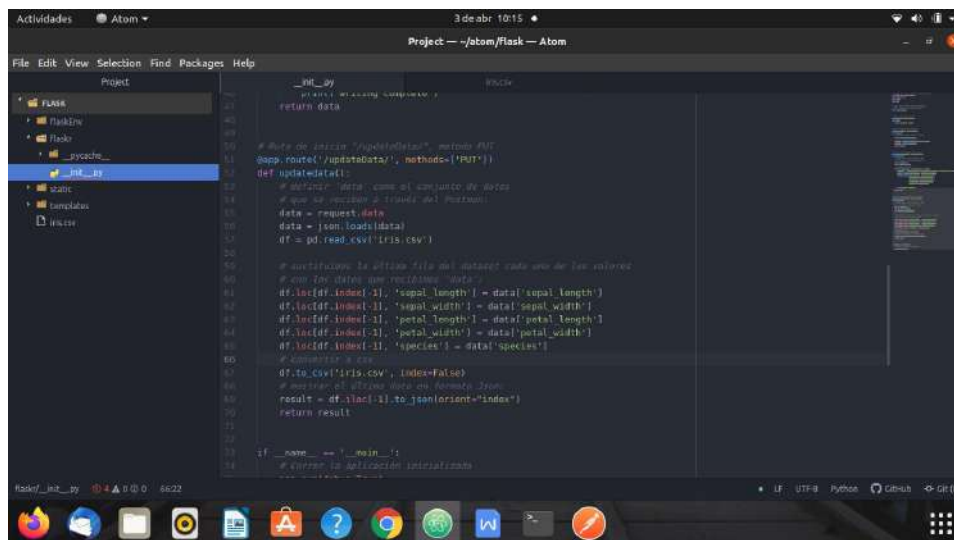


Figura 3.7. Archivo `__init__.py` donde se observa la programación del método PUT

Para probarlo vamos al Postman seleccionamos el método PUT `http://127.0.0.1:5000/updateData/` ponemos en el body el siguiente dato de prueba y damos a send:

Body --> raw --> json:

```
{ "sepal_length": 5.4, "sepal_width": 4.5, "petal_length": 3.2, "petal_width": 0.2, "species":  
"versicolor"}
```

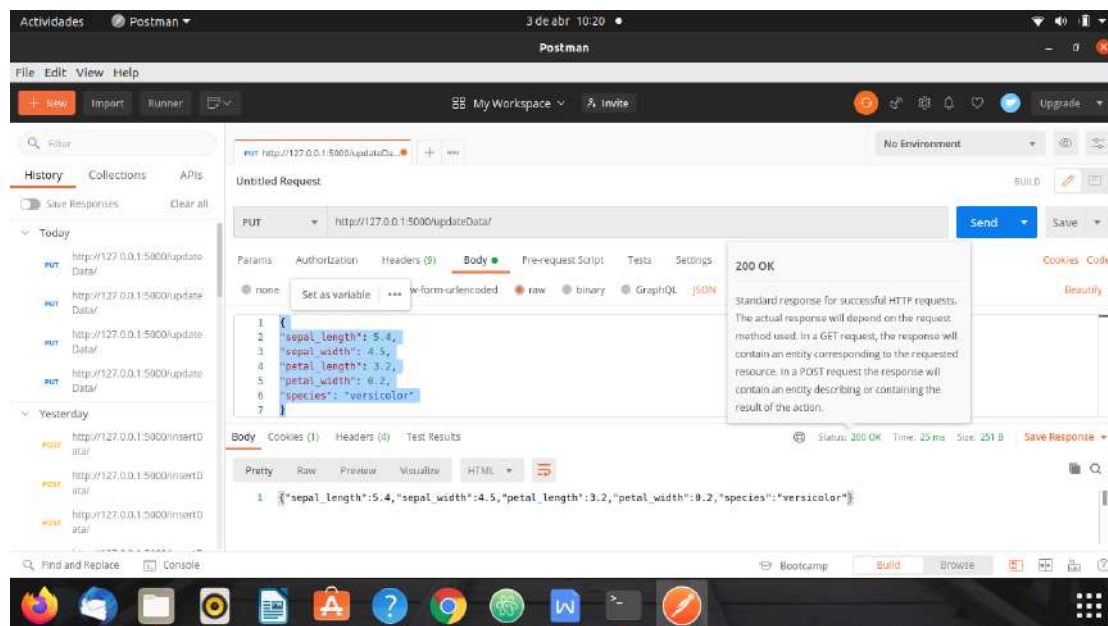


Figura 3.8. Método PUT realizado con Postman

Nos muestra un 200 Ok y nos muestra el último dato de nuestro dataset. Comprobamos en el archivo CSV que se ha actualizado correctamente:

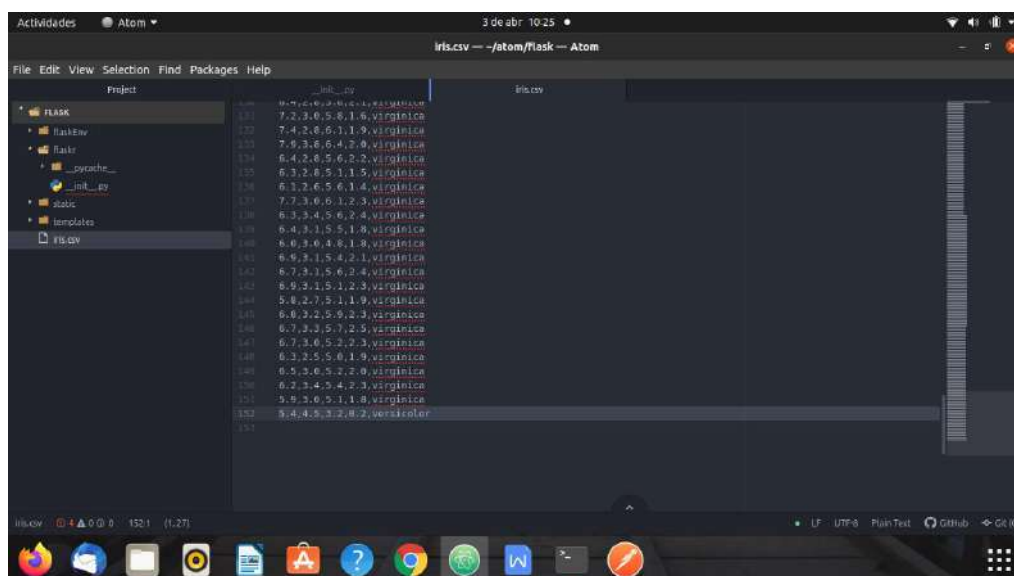


Figura 3.9. Archivo CSV donde vemos la actualización del último dato

3.4. Método DELETE

Definimos la función para el método DELETE:

```
# Ruta de inicio "/deleteData/", metodo DELETE
@app.route('/deleteData/', methods=['DELETE'])
def deleteData():
    df = pd.read_csv('iris.csv')
    # Eliminar la última fila
    df.drop(df.index[-1], inplace=True)
    # convertir a csv
    df.to_csv('iris.csv', index=False)
    # mostrar el último dato en formato Json:
    result = df.iloc[-1].to_json(orient="index")
    return result
```

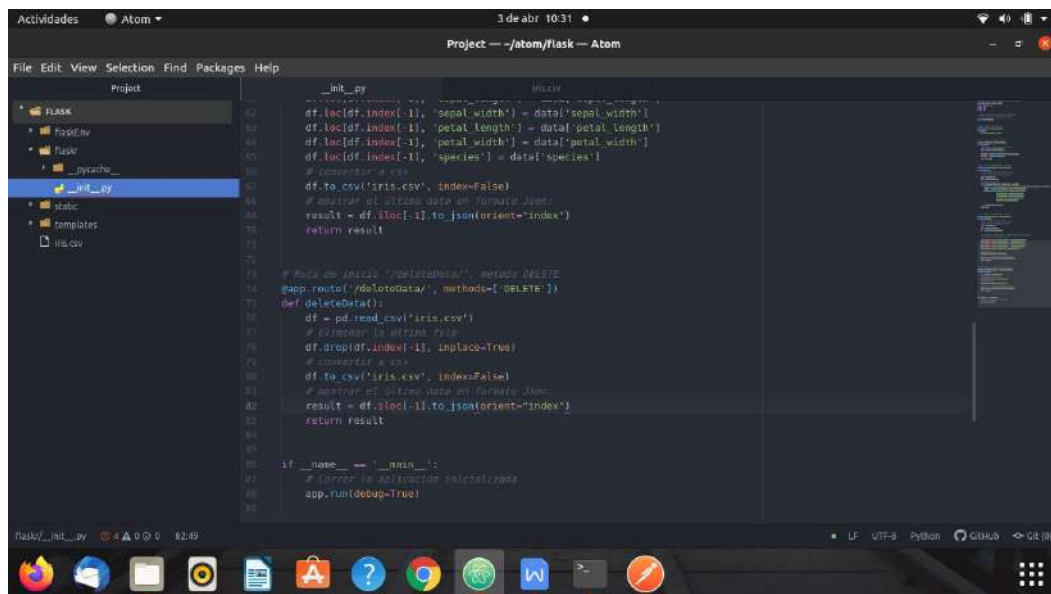


Figura 3.10. Archivo `__init__.py` donde se muestra el método DELETE

Para probarlo vamos al Postman seleccionamos el método DELETE `http://127.0.0.1:5000/deleteData/` ponemos en el body none y damos a send:

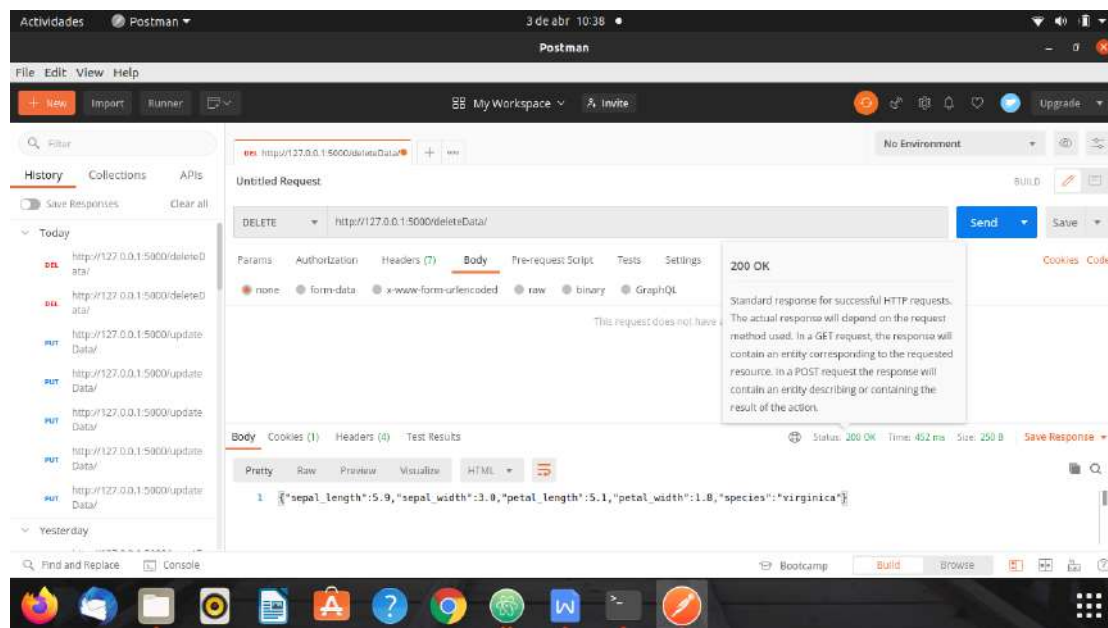


Figura 3.11. Método DELETE realizado en Postman

Nos da de resultado un 200 OK y nos muestra el último dato en el csv.

Para comprobarlo vamos el archivo CSV:

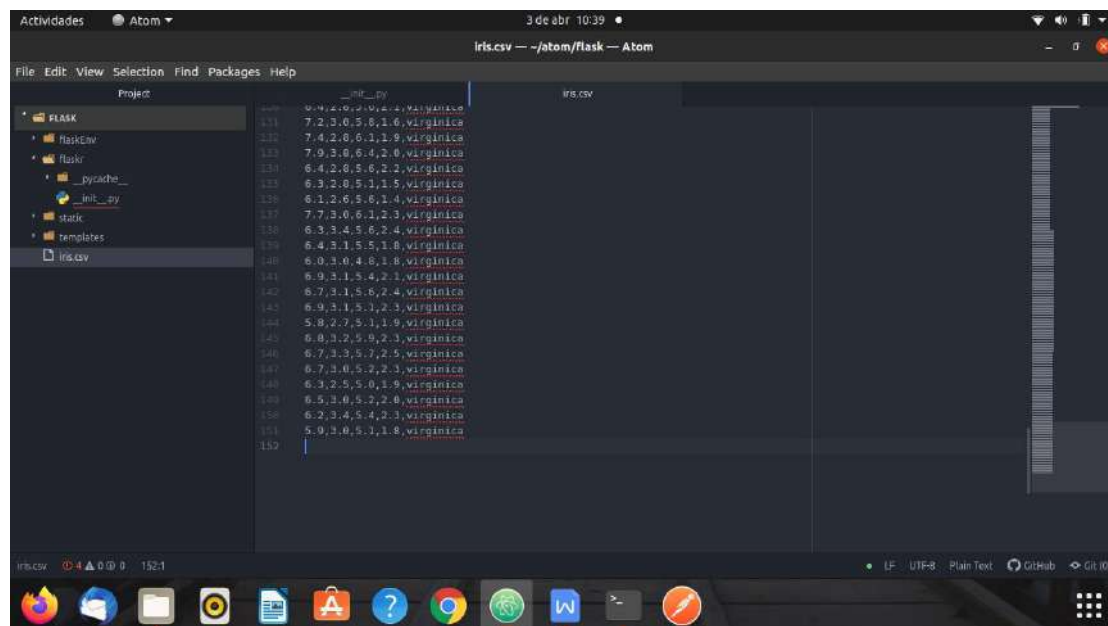


Figura 3.12. El archivo CSV donde se observa que se ha eliminado el último dato.

Observamos que se ha eliminado correctamente.

Nota:

Si probamos realizar un método que no hemos definido en la URL nos mostrará un error:

Si realizamos un POST a `http://127.0.0.1:5000/deleteData/` nos muestra un status code 405 método no permitido.

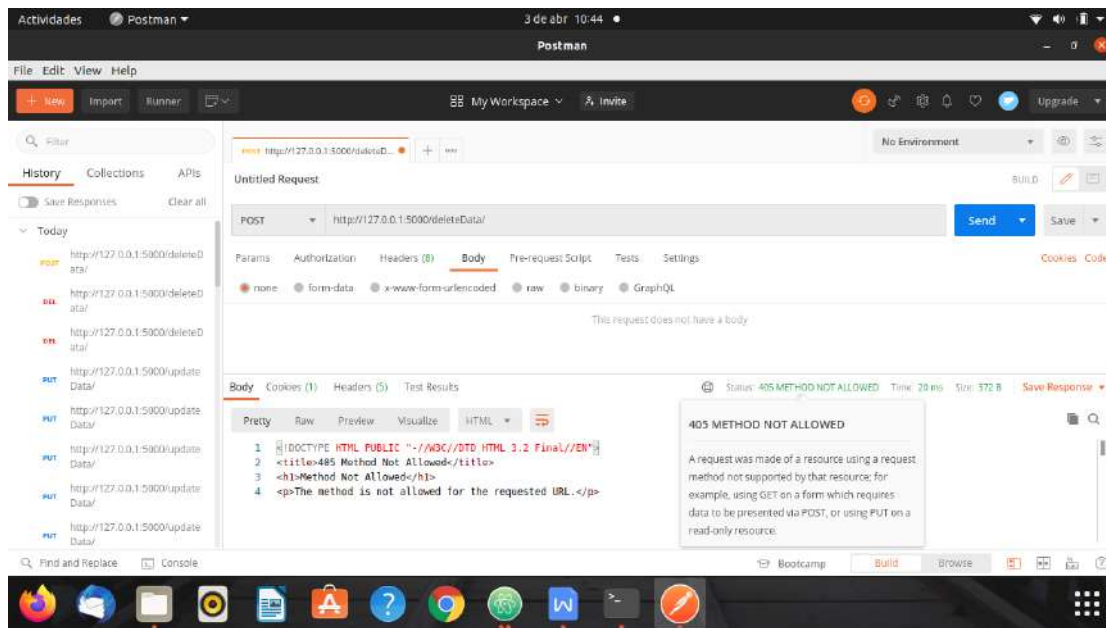


Figura 3.13. Error al realizar en la URL un método no permitido

4. PUNTOS CLAVE

- | Crea un entorno virtual siempre que empieces un proyecto.
- | En este caso hemos definido nuestros métodos en el archivo `__init__.py`
- | Postman es una herramienta que nos sirve para comprobar la funcionalidad de nuestra API REST.

